워크스테이션 클러스터 상에서 분산공유메모리 인터페이스로 배열 데이터의 공유를 지원하는 Java 패키지의 설계와 구현

임혜정[†] · 김 명[††]

요 약

본 연구에서는 배열 데이터를 여러 호스트 상에 분산시켜 생성하고 편리하게 공유할 수 있도록 하는 Java 패키지인 JPAS (Java Package for Array Sharing)를 설계하고 구현하였다. JPAS는 순수 Java로 구현되어 이식성이 뛰어나고, Java RMI를 이용하여 분산공유메모리 모델과 같이 위치 독립적인 접근 인터페이스로 배열 데이터를 공유할 수 있도록 한다. JPAS는 네트웍 오버헤드로 인한 성능 저하를 막기 위해서, 프로그래머가 알고 있는 애플리케이션의 특성을 공유 데이터 사용시에 반영할 수 있도록 한다. 또한, 데이터의 일관성을 유지하기 위해서, JPAS의 모든 배열 데이터들은 값을 갱신할 수 있는 메소드들을 갖는다. 실제로, 병렬 프로그램들을 작성하여 워크스테이션 클러스터 상에서 실행시켜 본 결과, JPAS가 비교적 우수한 성능의 병렬 프로그래밍 도구임을 보였다.

Design and Implementation of a Java Package for Sharing Array Data by the DSM Interface on a Cluster of Workstations

Haejung Lim[†] and Myung Kim[††]

ABSTRACT

In this paper, we present JPAS(Java Package for Array Sharing) which is a Java Package for sharing arrays of data on a cluster of workstations. It allows us to divide an array of data into several pieces, and to place each piece on a different host. JPAS uses Java RMI so that the entire array can be accessed by a location transparent interface which is similar to that of a distributed shared memory system. JPAS is portable and easy to use since it is implemented using pure Java.

In order to reduce network overhead, JPAS allows programmers to use their prior knowledge of the application. Data consistency can be maintained through the value updating methods defined for all the elements of an array. We developed parallel programs which use JPAS, and tested them on a cluster of workstations. The test results show that JPAS is a parallel programming tool with reasonably good performance.

1. 서 론

Java는 쓰레드를 기반으로 하여 병렬성과 동기화 기능을 제공하는 플랫폼 독립적인 언어이다. 특히, Sun사에서 제공하는 JDK(Java Development Kit) 1.1에는 객체 직렬화(object serialization)와 RMI(Remote Method Invocation) 기능이 포함되어 있다. 언어자체의 이러한 특성으로 인해, Java는 네트웍에 연결되어 있는 워크스테이션 클러스터 상에서 병렬처리할 때에 유용하게 사용될 수 있으며, 최근에는 Java를 기반으로 클러스터 상에서 병렬 컴퓨팅 환경을 구축하는 연구들이 활발하게 진행되고 있다[1,2,3,

† KIST 트라이볼로지센터 위촉연구원
†† 이화여자대학교 컴퓨터학과 조교수

8,10].

이러한 병렬컴퓨팅 환경은 크게 메시지 패싱 모델과 분산 공유 메모리(Distributed Shared Memory, 이하 DSM으로 약칭) 모델로 나뉜다. Java를 기반으로 메시지 패싱 모델을 구현한 예로는, JavaPVM[3], DPJ[2], JPVM[1], JPE[11]를 들 수 있다. 이들은 모두 메시지 패싱 방식이므로, 프로그래머가 데이터의 위치를 알고 있어야하고, 적절한 시점에 이들을 프로세서간에 이동시켜야 하는 불편함을 지닌다. Java를 이용하여 DSM 모델을 구현한 예로는, JavaParty[8]와 Java/DSM[10]을 들 수 있다. 이 Java DSM 시스템들은 일반적으로 DSM 모델의 특성상 표준 자바 가상 머신이외에 추가적인 기능을 제공하는 확장된 Java 가상머신이나 별도의 런타임 시스템을 실행시간에 필요로하여, 범용성이 떨어지는 단점이 있다.

프로그래머의 입장에서 본다면, 메시지 패싱 모델보다는 DSM 모델이 사용하기 편하다. 본 연구에서는 배열 데이터를 여러 호스트에 분산시켜 생성한 후, 편리하게 공유할 수 있도록 하는 JPAS (Java Package for Array Sharing)라는 Java 패키지를 설계하고 구현하였다. JPAS는 다음과 같은 3가지 특징을 갖는다. 첫째, 메시지 패싱 방식보다 편리한 DSM 모델의 인터페이스로 공유 데이터를 사용한다. 둘째, 성능 향상을 위해 프로그래머가 알고 있는 애플리케이션의 특성을 공유 데이터를 사용할 때 반영할 수 있도록 하여 불필요한 네트웍 오버헤드를 피한다. 셋째, 순수 Java로 구현되어 다른 플랫폼으로의 포팅이나 별도의 하드웨어나 소프트웨어 지원이 없어도 표준 JDK만 설치되면 모든 플래폼 상에서 사용할 수 있다.

JPAS는 DSM 모델처럼 공유 데이터의 위치 독립적인 접근 인터페이스를 제공하지만, DSM 모델과는 다음과 같은 차이가 있다. 일반적으로 DSM 시스템에서는 운영체제의 커널이나 런타임 시스템을 통해 가상적인 전역메모리를 제공함으로써 데이터의 물리적 위치에 독립적인 접근 인터페이스 뿐 아니라 공유 메모리와 캐쉬(cache)의 일관성도 자동적으로 유지 시켜준다. 반면에, JPAS에서는 운영체제나 런타임 시스템의 도움 없이 Java RMI를 이용하여 원격 객체가 가진 공유 데이터를 마치 로컬 데이터인 것처럼 접근할 수 있도록 하는 방식이다. 즉, 배열을 공유 데이터로 사용하는 경우, 이 배열을 Java 원격 객체(remote object)로 생성시켜 두고, 배열의 데이터를 사용하려는 호스트들은 자신의 로컬 객체의 메소드를 호출하는 것과 동일한 방식으로 배열 객체의 read(), write() 메소드를 호출하여 해당 데이터를 불러 쓸 수 있도록 한다. JPAS는 DSM 모델과 같이 시스템 내부적으로 데이터의 일관성(consistency)을 제공하지는 않지만, 배열 객체들이 최신 값으로 갱신될 수 있는 메소드들을 가지므로, 이를 이용하면 프로그램에서 필요로 하는 데이터 일관성을 유지할 수 있다. 또한, DSM 시스템들은 일반적으로 플랫폼 상의 설치(installation)과정을 필요로하는 반면에, JPAS는 Java 패키지 형태로 구현되었으므로 애플리케이션 프로그래밍 시에 JPAS 패키지를 임포트하여 프로그래밍한 후에 JPAS가 제공하는 Java 프로그램들과 함께 실행시키는 방식으로, 그 사용법이 간단하다.

JPAS는 배열 데이터의 타입으로 정수와 실수를 사용한다. 다른 데이터 타입이 현재 지원되지 않는 이유는 Java 가 type argument를 허용하지 않으므로 배열 데이터를 Java 원격 객체로 생성할 때 그 배열의 다양한 원소 타입을 바디 생성 프로그램(2.2절 참조)에게 파라미터로 넘겨줄 수 없기 때문이다. 최근에는 C++의 템플릿(template) 같은 type parameterization 기능을 Java에 추가하는 방법[7,9]들이 연구되고 있는데, 이를 JPAS와 연계시키면 Java 객체나 사용자 정의 데이터 타입도 배열의 원소로 사용될 수 있게 되므로, JPAS는 보다 일반적인 집합적(enumerated) 데이터의 공유를 지원하는 Java 패키지로 확장될 수 있다.

본 논문의 구성은 다음과 같다. 2장에서 JPAS의 전체적인 구성 및 구현 방법을 소개한다. 3장에서는 예제 프로그램을 통해 JPAS API를 살펴보고, 4장에서는 JPAS를 사용한 2개의 예제 프로그램을 워크스테이션 클러스터에서 실행시켜 성능을 분석한다. 5장에서는 본 연구에서 얻은 결과를 분석하고 향후 연구과제에 대해 정리한다.

2. JPAS의 설계와 구현

우선 클러스터내의 여러 워크스테이션에 분산되어 있는 배열 데이터를 JPAS을 사용하여 공유하는 방법을 설명한다. 공유 데이터의 구조, 생성, 접근 방식과 특징이 여기서 언급된다. 그 다음에는 JPAS의

시스템 구조와 구현 방식을 소개하기로 한다.

2.1 JPAS의 공유 데이터 구조

JPAS는 메시지 패싱 모델이 갖는 단점인 프로그래머가 데이터의 위치를 기억하고, 적절한 시점에 프로세서간에 이들을 이동시켜야 하는 불편을 덜기 위해, DSM 모델과 같은 프로그래밍 인터페이스를 제공한다. 클러스터내의 프로세서들간에 공유할 데이터는 그림 1과 같은 복합 객체 (composite object) [4,5] 형태로 구성된다. 복합 객체는 하나의 핸들(handle)과 여러 개의 바디(body)로 이루어진 객체이며, 핸들과 바디는 각각 객체 타입(object type)으로 구현된다. 바디는 공유 데이터의 실제 값을 저장하고, 이 데이터의 원소를 다룰 수 있는 기본(primitive) 연산을 멤버 메소드로 갖는다.

핸들은 다음 3가지 기능을 통해 데이터와 사용자 사이의 인터페이스를 담당한다. 첫째, 공유 데이터를 저장하는 바디들을 생성한다. 둘째, 바디에 관한 정보를 관리한다. 복합 객체 구조에서 바디와 핸들은 서로 다른 주소 공간에 분산되어 존재할 수 있다. 그러므로 핸들은 자신이 생성한 바디의 실제 위치가 어디인지, 공유 데이터의 각 원소는 어느 바디에 속해 있는지 등의 정보를 Partition과 Directory라는 자료구조에 저장해 두고 관리한다. 셋째, 바디가 다른 주소 공간에 존재할 수 있으므로, 핸들은 적절한 통신수단을 통해 원격 바디에 접근하고 저장된 데이터를 사용할 수 있도록 메소드를 제공한다. 즉, 핸들이란 공유 데이터를 대표하는 객체이다. 실제로 모든

그림 1. 복합 객체 구조

공유 데이터는 바디에 저장되어 여러 호스트 상에 분산되어 존재하고, 복합 객체 구조로 캡슐화되어 핸들의 메소드로만 접근된다.

이러한 복합 객체 구조는 다음과 같은 특징을 갖는다. 첫째, 그림 1의 예에서와 같이 프로세서 1의 핸들이 프로세서 4 에 복사되어 전달되면, 이 두 프로세서는 서로 바디를 공유하는 효과를 얻는다. 둘째, 하나의 데이터는 여러 개의 바디들로 나뉘어 저장될 수 있고, 나뉘어진 바디들은 클러스터 상에 분산되어 위치할 수 있다. 이렇게 개념적으로는 하나인 데이터를 여러 개의 독립적인 바디로 나누어 사용하면, 한 데이터를 동시에 "다중 읽기(multiple read)"를 하거나 "다중 쓰기(multiple write)"를 할 수 있다. 셋째, 사용자가 알고 있는 데이터의 용도와 위치 정보를 반영한 메소드를 핸들에 구현하여 성능 좋은 데이터 공유를 할 수 있다. 복합 객체 구조에서 바디는 원시 연산 메소드을 가지고 분산되어 존재하므로, 핸들은 Partition/Directory를 통해 원하는 바디를 찾아 네트웍으로 접근한다. 그리고, 핸들은 바디가 가진 원시 연산 메소드를 조립(assemble)하여 프로그래머가 원하는 상위 수준 메소드를 제공한다. 이 때, 복합 객체 구조에서는 그림 1처럼 일반 핸들이 가진 Partition과 Directory 정보는 그대로 사용하면서 바디의 원시 연산 메소드를 다른 방식으로 조합하여 특수한 기능을 제공하는 확장 핸들로 교체할 수 있도록 한다. 확장 핸들로 교체되고 나면 애플리케이션에서는 이 확장 핸들의 특수한 메소드를 통해 공유 데이터를 사용할 수 있으므로, 여전히 DSM 인터페이스로 공유 데이터를 접근할 수 있을 뿐만 아니라 불필요한 네트웍 오버헤드를 개선하여 성능을 향상시킬 수 있다.

2.2 JPAS 시스템 개요 및 구현

JPAS는 2.1에서 설명한 복합 객체 구조를 바탕으로 Java RMI와 소켓 통신을 통해 DSM 인터페이스를 지원한다. JPAS는 표1과 같이 공유 데이터 생성에 사용되는 Java 클래스들과 애플리케이션의 실행 환경을 만들어주는 2개의 Java 프로그램으로 이루어진다. 이 프로그램들과 클래스들은 그림 2~그림 8과 같이 여러 프로세서들 상에 공유 데이터를 생성, 분산, 공유하는데 사용된다.

Java는 다른 주소공간에, 즉 다른 Java 가상 머신
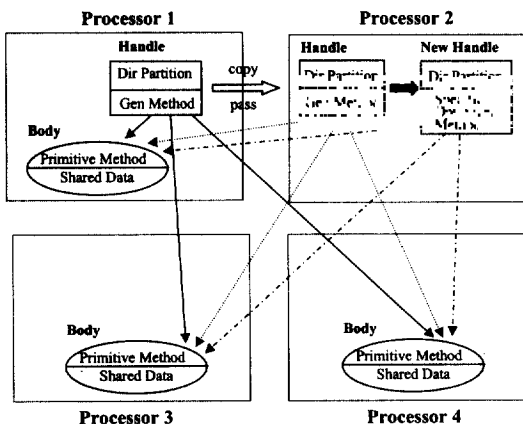
표 1. JPAS 패키지의 구성

Java 프로그램	Java 클래스	
	공유 데이터용 클래스	기타 클래스
BodyCreator : BodyCreator.java	[illegible] : Body_i.java - 정수용 데이터 : Body_f.java - 실수용 데이터	[illegible] : JPASApp.java
	[illegible] : Handle_i.java - 정수용 핸들 : Handle_f.java - 실수용 핸들	
[illegible] : BarrierManager.java	[illegible] : Cache_i.java, Buffer_i.java - 정수용 캐쉬와 버퍼 : Cache_f.java, Buffer_f.java - 실수용 캐쉬와 버퍼	[illegible] : Barrier.java

안에, 독자적으로 객체를 생성시키는 기능을 가지고 있지 않다. 그러므로 핸들이 독립적인 여러 호스트 상에 바디를 만들어 복합 객체를 구성하기 위해서는 각 호스트 상에 바디를 생성해주는 프로그램이 필요하다. 이 프로그램을 바디 생성(Body Creator) 프로그램이라고 부른다. 바디 생성 프로그램은 Java 원격 객체로 구현되어 JPAS 애플리케이션이 실행될 때 백그라운드 프로세스로 함께 실행된다. 이때 자신의 server 레퍼런스(skeleton)를 RMI 레지스트리(registry)에 등록(그림2의 ① 참조)하게 된다. 바디 생성 프로그램이 레지스트리에 등록된 후, JPAS 애플리케이션에서 공유 데이터를 생성하려면 먼저 BCInit()라는 메소드를 호출(그림 3의 ② 참조)하여 모든 호스트의 레지스트리에 등록된 바디 생성 프로그램의 client 레퍼런스(stub)를 찾아(그림 3의 ③ 참조) BCR (Body Creator Reference) 테이블에 저장

그림 2. 바디 생성 프로그램의 실행
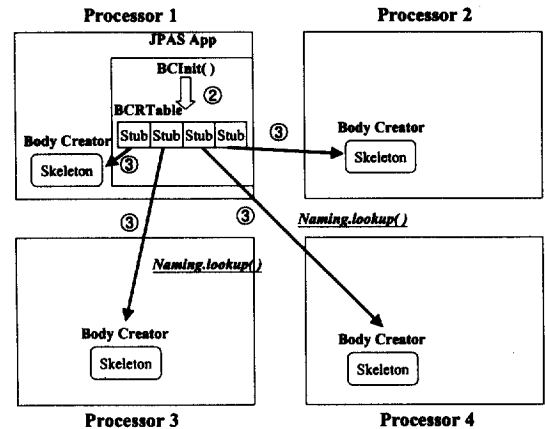
그림 3. BCR 테이블의 초기화

한다.

BCR 테이블에서 바디를 생성시킬 호스트들의 바디 생성 프로그램의 레퍼런스를 추출하여 배열의 가로, 세로 크기 및 분할 정보(가로 분할 또는 세로 분할)와 함께 핸들 생성자에게 파라미터로 넘겨주면(그림 4의 ④ 참조) 핸들 생성자는 이를 이용하여 Partition 함수를 초기화 한 후, 원격 호스트의 바디 생성프로그램들에게 RMI로 바디 생성을 요청(그림 4의 ⑤ 참조)한다. 요청을 받은 바디 생성 프로그램은 바디 역시 Java 원격 객체로 생성, 바디의 server 레퍼런스를 레지스트리에 등록(그림 5의 ⑥ 참조)한 후, client 레퍼런스를 핸들에게 넘겨준다(그림 5의 ⑦ 참조).

Partition 함수는 배열의 분할 정보를 유지하는 함수로서 배열이 가로 또는 세로로 분할되어 생성되었는지 따라 배열의 각 원소가 저장되어 있는 바디의 번호를 리턴해준다. 현재 JPAS에서는 구현상의 편
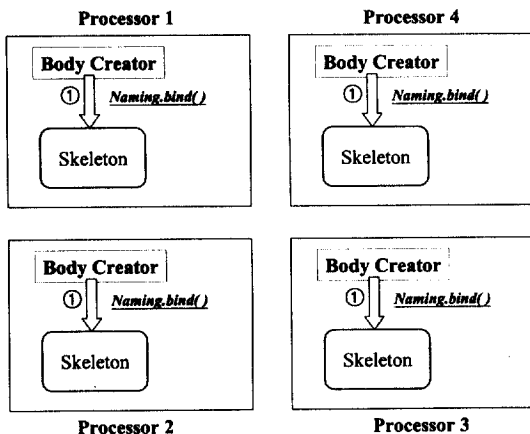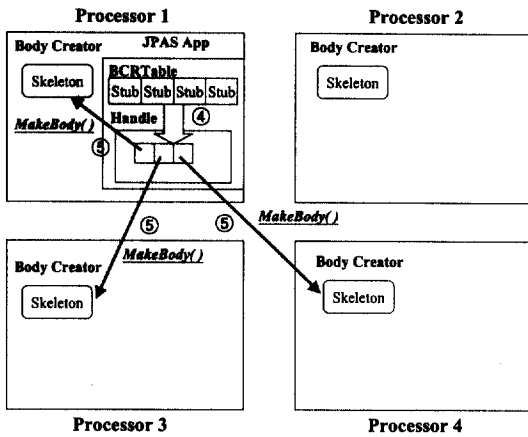
그림 4. RMI를 이용한 바디 생성
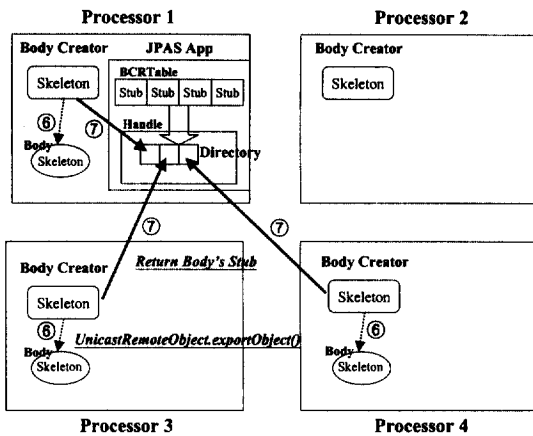
그림 5. 공유 데이터의 완성

의를 위해 배열의 분할 방식으로 가로 분할과 세로 분할 방식을 제공한다. Partition 함수의 배열 원소 인덱스(index)를 계산하는 알고리즘을 바꿔 주면 순환적 분할(cyclic)또는 블록 단위의 분할(block) 방식 등 다양한 분할 방식도 구현 가능하다.

바디 생성 프로그램이 바디를 생성한 후, client 레퍼런스를 보내주면(그림 5의 ⑦ 참조) 핸들은 이를 받아 Directory에 저장하여 두고 바디에 저장된 공유 데이터를 접근할 때는 Partition 함수와 Directory를 이용하여 원하는 데이터를 찾아가도록 한다(그림 6의 ⑧ 참조). 예를 들어, 배열의 i 행 j 열 원소가 필요한 경우, Partition함수를 통해 이 원소가 어느 바디에 저장되어 있는지 알아낸 후 Directory에서 이 바디의 레퍼런스인 Stub을 얻어서 RMI로 접근하면 된다. 이렇게 한 프로세스에서 만든 핸들을 소켓을 통해 다른 프로세스에게 전달(그림 7의 ⑨ 참조)하게 되면 그 프로세스 역시 받은 핸들을 통해 바디의 데이터를 접근(그림 7의 ⑩ 참조)할 수 있게 된다.
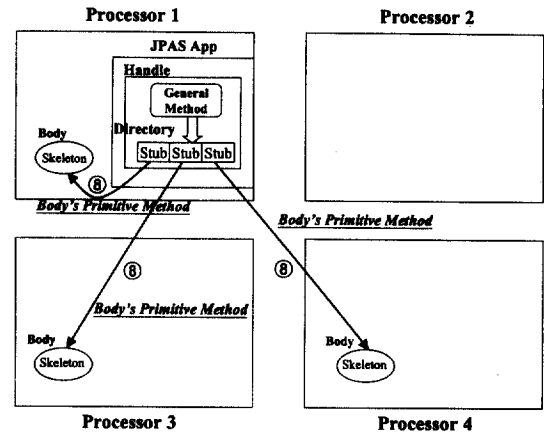
그림 6. RMI를 이용한 원격 데이터 접근
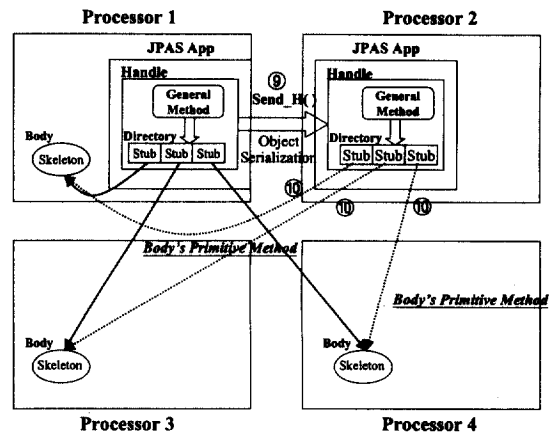
그림 7. 핸들을 이용한 데이터 공유

지금까지 설명한 핸들은 RMI를 통해 원격 바디를 즉시 접근한 후, 바디의 원시 연산을 이용하여 공유 데이터를 읽고 쓰는 일반적 메소드를 가진다. 그러므로 이런 일반 핸들이 제공하는 메소드보다 특수한 메소드를 사용하려고 할 때는 새로운 메소드를 갖는 별도의 핸들 클래스를 미리 구현한 후(그림 8의 ⑪ 참조), Directory/Partition 정보는 일반 핸들의 것을 그대로 받아서 사용하면(그림 8의 ⑫ 참조) 애플리케이션에서 필요로 하는 특수한 확장 핸들을 쉽게 만들어 쓸 수 있다.
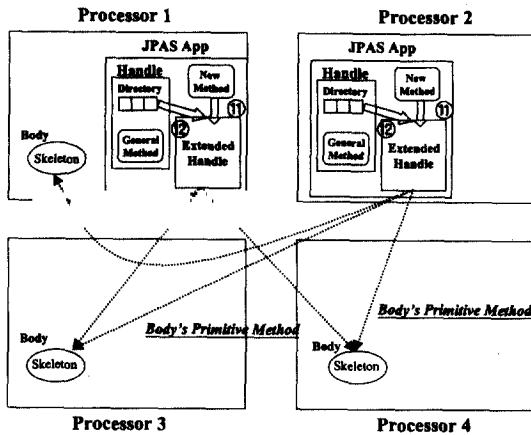
그림 8. 확장 핸들의 생성 및 사용

이 외에도 JPAS에서는 여러 프로세스간의 작업 동기화를 위해 Barrier 클래스와 BarrierManager를 제공한다. BarrierManager는 백그라운드 프로세스로 실행되어 다른 프로세스들이 보내는 동기화 신호를 관리하는 Java 프로그램이다. 애플리케이션과 BarrierManager를 함께 실행시키면 BarrierManager는 실행 시작과 함께 Java의 서버 소켓(ServerSoket)을 열고 프로세스들로부터 들어오는 Barrier 도착 신호(sync())를 기다린다. 그리고 애플리케이션에서 동기화가 필요할 때 Barrier 클래스의 sync() 메소드를 호출하면 소켓을 통해 BarrierManager에게 Barrier 도착을 알리게 된다. 작업에 참여하는 모든 프로세스들이 sync() 메소드를 통해 Barrier 도착 신호를 보내면 BarrierManager 역시 Java의 소켓을 통해 모든 프로세스에게 작업 재시작을 알려서 프로세스간의 작업을 동기화 시킨다.

3. JPAS API와 예제 프로그램

본 절에서는 예제 프로그램을 통해 JPAS의 사용자 API를 설명하고자 한다. JPAS는 master-slave모드나 SPMD 모드로 사용될 수 있는데, 여기서는 master-slave 모드로 작성된 JPAS 행렬 곱셈 프로그램을 소개한다.

$n \times n$ 행렬 A와 B를 곱하여 행렬 C를 구할 때 $C[i, j]$는 다음과 같이 계산된다.: $C[i,j]=A[i,0]*B[0,j]+A[i,1]*B[1,j]+\cdots+A[i,n-1]*B[n-1,j]$. 본 예제에서는 행렬 A와 C를 그림 4와 같이 분할하여 각 블록을 하나의 바디로 생성한 후 클러스터 상의 여러 프로세스들이 행렬 A의 한 블록과 행렬 B 전체를 곱하여 행렬 C의 한 블록을 계산하도록 하였다.
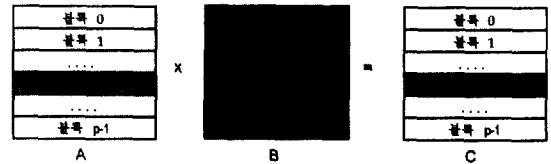
그림 9. 행렬 곱셈을 위한 행렬 분할

JPAS을 이용할 때는 우선 JPAS 패키지를 임포트한 후, JPASApp 클래스를 상속하여 애플리케이션을 작성한다. JPASApp는 JPAS 애플리케이션을 작성하는데 템플릿 역할을 하는 클래스인데, 이를 상속하면 main() 메소드에서 JPASApp클래스가 제공하는 여러 가지 메소드를 이용하여 JPAS 애플리케이션을 작성할 수 있다.

master 프로그램은 공유 데이터를 복합 객체 구조로 생성한 후 핸들을 slave 프로그램에게 보내주어, 각 slave 프로그램들이 자신에게 할당된 연산을 할 수 있도록 한다. 행렬 곱셈 문제의 master 프로그램은 다음과 같다.

master 프로그램은 공유 데이터를 생성하기 전에 먼저 BCInit()(4행)을 실행하여 모든 호스트상의 BodyCreator 클라이언트 레퍼런스를 찾아 JPASApp 내의 BCR 테이블에 저장한다. 그리고 행렬 A, B, C의 바디가 생성될 호스트 번호를 배열 temp1, temp2. temp3 에 각각 명시한 후(5행), JPASApp 클래스의 MkList() 메소드를 통해 BCR 테이블에서 이 호스트들만의 BC 레퍼런스를 추출하여 list1, list2, list3을 구성한다(6행). list1, list2, list3과 행렬의 가로, 세로 크기, 그리고 분할 정보(0=가로분할,1=세로분할)를 파라미터로 복합 객체 생성자를 호출하면 행렬 A, B, C가 복합 객체 형태로 생성된다(8-10행). 이렇게 공유 데이터를 생성한 후 SendH_i() 메소드를 통해 배열 slaves에 명시된(11행) 호스트들에게 행렬들의 핸들 A, B, C를 전송한다(12행). slave 프로그램은 그림 11에 있다.

우선, 모든 slave 프로그램은 JPASApp 클래스의 MyRank() 메소드를 호출하여 자신이 실행되고 있

```
1  [illegible]
2  public class Master extends JPASApp {
3    public static void main(String args[ ]) {
4       [illegible]
5       int[ ] temp1 = {0, 1, 2, 3}; int[ ] temp2 = {0};  int[ ] temp3 = {0, 1, 2, 3};
6       IBC[ ] list1 = MkList(temp1); IBC[ ] list2 = MkList(temp2); IBC[ ] list3 = MkList(temp3);
7       int mxrow = 4;    int mxcol = 4;
8       H_i A = new H_i(list1, mxrow, mxcol, 0);
9       H_i B = new H_i(list2, mxrow, mxcol, 0);
10      H_i C = new H_i(list3, mxrow, mxcol, 0);                    : JPAS API
11      int[ ] slaves = {1, 2, 3};
12      SendH_i(A, slaves);  SendH_i(B, slaves);   SendH_i(C, slaves);
13   }
14 }
```

그림 10. 행렬 곱셈 문제의 master 프로그램

```
1  import JPAS;
2  public class Slave extends JPASApp {
3      public static void main( String args[ ] ) {
4      int mytask = MyRank( );
5      H_i A = new H_i( );   H_i B = new H_i( );   H_i C = new H_i( );
6      RecvH_i(A, B, C);
7      Cache_i xA = new Cache_i(A, mytask);   Cache_i xB = new Cache_i(B, 0);
8      Buffer_i xC = new Buffer_i(C, mytask);
9      int i, j, k, sum;
10     for(i = 0; i < xA.CacheRow; i++)
11          for(j = 0; j < xB.CacheCol; j++){
12               sum = 0;                                      : JPAS API
13               for(k = 0; k < xB.CacheRow; k++)
14                    sum = sum + xA.Read(i, k) * xB.Read(k, j);
15               xC.Write(i, j, sum);
16          }
17      xC.Flush( );
18     }
19 }
```

그림 11. 행렬 곱셈의 slave 프로그램

는 호스트의 번호를 구한다. 이 번호는 나중에 각 slave가 담당하는 행렬의 블록 번호를 찾는데 사용된다(7, 8행). 그 다음, 각 slave 프로그램은 핸들을 받기 위해 비어있는 핸들을 먼저 생성하고(5행) RecvH_i() 메소드를 통해 master 프로그램이 보낸 핸들을 이곳에 받아 저장한다(6행). master가 보내준 이 핸들 A, B, C는 원격 호스트에 존재하는 바디에 대해 항상 RMI로 즉시 접근, 읽고 쓰는 일반적인 메소드를 제공한다. 행렬 곱셈 문제에서 행렬 A와 B는 읽기 연산만 자주 사용하는 데이터이므로 매번 일반 핸들의 RMI 메소드로 접근하기보다는 로컬 메모리의 읽기용 캐쉬에 저장해 두고 사용하면 불필요한 네트웍 오버헤드를 피할 수 있다. 행렬 C 역시 매번 RMI로 접근하여 값을 읽고 쓰기보다는 로컬 버퍼에 곱셈

결과 값을 저장해 두었다가 모든 계산이 끝난 후 실제 행렬 *C*에 한꺼번에 갱신하는 것이 좋다.

JPAS에서는 이런 경우를 위해 일반 핸들의 메소드 일부를 오버로드(overload)한 확장 핸들인 Cache와 Buffer 클래스를 제공한다. 이는 일반 핸들과 유사하나 읽기와 쓰기 메소드가 각각 로컬 버퍼와 캐쉬를 이용하도록 오버로드한 핸들이다. master가 보내준 일반 핸들 *A*, *B*, *C*를 파라미터로 확장 핸들 생성자를 호출하면(7, 8행) RMI로 원격 데이터를 직접 접근하는 대신 로컬 메모리에 저장된 데이터를 읽고 쓰면서 일반 핸들 *A*, *B*, *C*가 가진 Directory/Partition 정보는 그대로 받아 사용할 수 있는 확장 핸들이 생성된다. 이렇게 확장 핸들을 생성하고 나면 for 루프 안에서의 모든 곱셈 연산 시에는 확장 핸들인 Cache의 read 메소드를 통해 행렬 값을 읽고 곱셈의 결과 값도 역시 확장 핸들인 Buffer의 write() 메소드를 사용하여 로컬 버퍼에 기록해 두었다가 모든 계산이 끝난 후 flush() 메소드를 통해 실제 행렬 C에 한번에 갱신한다.

JPAS의 특성 중의 하나는 애플리케이션에 따라 이처럼 유용한 확장 핸들을 사용자가 직접 구현하여 사용할 수 있다는 점이다. 즉, JPAS에서 제공하는 핸들 클래스를 상속하되 필요한 메소드를 추가, 또는 오버로드하는 확장 핸들 클래스를 만들어 놓고 일반 핸들을 파라미터로 확장 핸들을 생성하면 사용자가 정의한 메소드도 JPAS의 일반 핸들 메소드처럼 DSM 인터페이스로 사용될 수 있다.

언뜻 보면 JPAS의 master와 slave 간에 핸들을 주고 받는 과정이 마치 메시지 패싱 모델에서 메시지를 주고받는 것과 같은 방식으로 보일 수 있다. 그러나 메시지 패싱 모델에서는 데이터를 사용할 때마다 계속적으로 프로세스간에 메시지 패싱을 통해 공유 데이터를 얻어야하는 반면, JPAS 프로그램에서는 연산에 들어가기 전에 공유 데이터에 대한 핸들을 얻고 나면 이 핸들을 통해 언제든지 공유 데이터를 접근할 수 있으므로 메시지 패싱에서 데이터를 공유하는 방법과는 다르다.

4. JPAS의 성능 평가

JPAS의 공유 데이터 접근 인터페이스와 동기화 기능을 이용해서 실행될 수 있는 행렬을 사용하는 병렬 프로그램을 작성하여 JPAS 성능을 측정, 분석해 보았다. 실험은 10Mbps 이더넷으로 연결된 워크스테이션 클러스터에서 수행하였다. 클러스터의 각 프로세서는 143MHz의 UltraSparc으로, 64MB의 메모리를 가지며 Solaris 2.5.1과 JDK 1.1. Java 가상머신을 탑재하고 있다. 병렬 프로그램을 1개의 프로세서에서 실행시킨 경우와 여러 개의 (2-8개) 프로세서를 써서 실행시킨 경우의 실행 시간을 측정하여 상대적 속도 향상(speedup)을 비교하였다. 정확한 결과를 얻기 위해 각 실험을 10회 정도 반복, 평균을 내어 속도 향상을 구하였다.

4.1 행렬 곱셈 프로그램

3장에서 예제로 소개한 행렬 곱셈 실험에는 512×512 정수행렬과 1024×1024 정수행렬을 사용하였다. 성능 측정 결과는 표 2와 그림 12에 있다. 표 2를 보면 행렬 곱셈 프로그램 실행에 걸린 전체 시간 중 캐쉬 읽기 시간(slave 프로세스가 행렬 *B*와 자신이 맡은 행렬 *A*의 한 블록을 로컬 캐쉬로 읽어오는데 드는 시간)이 크다는 것을 알 수 있다. 이는 주로 핸들의 RMI 메소드 사용에서 기인하는 오버헤드라고 볼 수 있다. Java RMI는 객체 타입 정보를 넘겨주고 처리하는 과정을 실행 시간에 이루어지도록 하므로 유연성을 가지나, 이러한 유연성을 위해 심각한 실행시간 오버헤드를 야기하기도 한다. 최근 연구[6]에 의하면 실행시간에 이루어지던 대부분의 객체 타입 정보의 처리를 컴파일 시간에 함으로써 RMI 고유의 다형성과 interoperability는 그대로 살리면서 기존의 RPC에 견줄 만한 성능을 낼 수 있는 RMI 기법이

표 2. JPAS를 이용한 행렬 곱셈 문제의 실험 결과

512×512 정수 행렬			1024×1024 정수 행렬		
4 PEs	캐쉬 읽기 시간	90 초	4 PEs	캐쉬 읽기 시간	640초
	곱셈 시간	150 초		곱셈 시간	1260 초
	기타	30초 이하		기타	80초 이하
	총 실행 시간	270초		총 실행 시간	1980 초
8 PEs	캐쉬 읽기 시간	80 초	8 PEs	캐쉬 읽기 시간	450 초
	곱셈 시간	85 초		곱셈 시간	630 초
	기타	15초 이하		기타	45초 이하
	총 실행 시간	180 초		총 실행 시간	1125 초
순차 프로그램 실행시간 : 734 초			순차 프로그램 실행시간 : 5930 초		
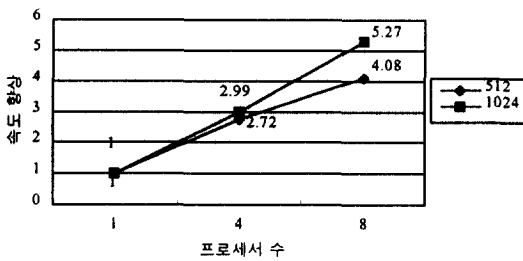
그림 12. 행렬 곱셈 프로그램의 성능 향상(n=512, 1024인 경우)

개발되고 있다고 한다. 이런 새로운 RMI 기법을 JPAS에 적용하면 현재 RMI를 주된 통신 수단으로 이용함으로 겪는 성능 저하 현상을 크게 개선할 수 있으리라고 본다.

4.2 2-D diffusion

2-D diffusion 문제는 어떤 물질이 2차원 공간 내에 산포되어 있을 때 그 물질의 산포되어 있는 정도(산포도)를 구하는 문제로, 대표적인 병렬 프로그램 중에 하나이다. 본 실험에서는 JPAS의 성능을 측정하기 위해 $n \times n$ 행렬의 각 원소가 한 점에서의 밀도 값을 나타낸다고 가정하고 이웃한 8개 원소 값들의 평균을 구하여 이 값을 그 점에서의 새로운 밀도 값으로 갱신하여 2-D diffusion 문제를 시뮬레이션 하였다. 비교적 정확한 산포도를 구하기 위해서 전체 행렬에 대해 50회, 100회 반복(iterate)해서 계산하였다.
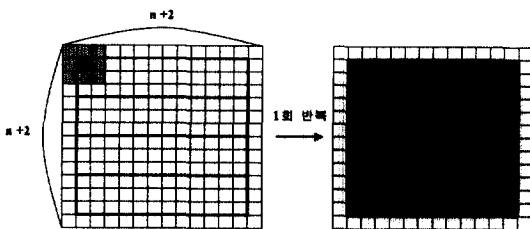
그림 13. JPAS에서의 2-D diffusion 계산

JPAS을 이용하는 2-D diffusion 프로그램에서는 먼저 프로세스 0 가 $(n+2)\times(n+2)$ 실수 행렬을 복합 객체 형태로 생성한다. 이 행렬은 그림 13에서와 같이 가로로 동등하게 분할된 여러 개의 블록으로 이루어지고 각 프로세스는 이 블록을 하나씩 맡아 로컬 캐쉬로 가져가 diffusion을 계산한다. 각 프로세스는 자신이 맡은 블록의 계산을 마치고 나서 다음 반복을 시작하기 위해서는 이웃 프로세스가 계산한 새로운 경계행 값을 필요로 한다. 그러므로, 모든 프로세스는 자신의 계산이 끝난 후에도 다른 프로세스들의 작업이 끝날 때까지 기다렸다가 이웃 블록의 새로운 경계 행 값을 얻어서 다음 반복을 시작해야 한다. 2-D diffusion 문제에서는 이러한 프로세스간의 작업 동기화와 데이터 공유를 위해 JPAS의 Barrier 클래스와 데이터를 특수하게 이동해주는 확장 핸들을 만들어 사용하였다. 2-D diffusion 문제의 실험 결과는 표3과 그림 14, 15에 나와있다.

표 3을 보면 프로세서의 수가 증가할수록 산포도를 구하는 계산 차제의 시간은 감소하지만 매 반복(iteration) 후에 Barrier 클래스를 통한 동기화와 행렬 블록의 경계 행들을 주고 받는데 드는 네트웍 오버헤드로 인해 속도 향상이 둔화됨을 알 수 있다. 행렬 곱셈 문제의 경우 대부분의 오버헤드가 Java RMI로 인한 것인데 반해 2-D diffusion에서는 Barrier 클래스가 이용하는 소켓 통신이 성능을 결정하는 주요 요소이므로 실험 환경의 네트웍 성능이 실험 결과에
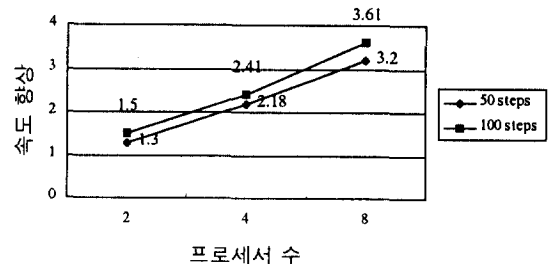
그림 14. 512×512 2-D diffusion 프로그램의 성능 향상
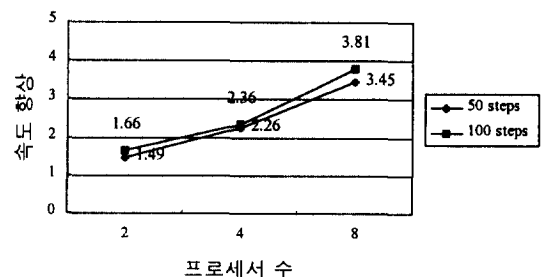
그림 15. 1024×1024 2-D diffusion 프로그램의 성능 향상

표 3. JPAS을 이용한 2-D diffusion 문제의 실험 결과

512×512 실수 행렬				1024×1024 실수 행렬			
2 PEs	50회	계산시간	165 초	2 PEs	50회	계산시간	675 초
		네트웍 오버헤드	85 초			네트웍 오버헤드	205 초
		총 실행시간	250 초			총 실행시간	880 초
	100회	계산시간	330 초		100회	계산시간	1350 초
		네트웍 오버헤드	104 초			네트웍 오버헤드	210 초
		총 실행시간	434 초			총 실행시간	1560 초
4 PEs	50회	계산시간	90 초	4 PEs	50회	계산시간	335 초
		네트웍 오버헤드	60 초			네트웍 오버헤드	245 초
		총 실행시간	150 초			총 실행시간	580 초
	100회	계산시간	180 초		100회	계산시간	670 초
		네트웍 오버헤드	90 초			네트웍 오버헤드	430 초
		총 실행시간	270 초			총 실행시간	1100 초
8 PEs	50회	계산시간	47.5 초	8 PEs	50회	계산시간	175 초
		네트웍 오버헤드	54.5 초			네트웍 오버헤드	205 초
		총 실행시간	102 초			총 실행시간	380 초
	100회	계산시간	95 초		100회	계산시간	350 초
		네트웍 오버헤드	85 초			네트웍 오버헤드	330 초
		총 실행시간	180 초			총 실행시간	680 초
순차	50회	총 실행시간 : 326.8 초		순차	50회	총 실행시간 : 1312 초	
	100회	총 실행시간 : 650.5 초			100회	총 실행시간 : 2597 초	

큰 영향을 준다고 할 수 있다. 본 실험이 10 Mbps 이더넷 환경에서 이루어진 점을 고려할 때 보다 고속의 네트웍 환경에서 사용한다면 소켓 통신비용이 감소하여 JPAS의 성능이 훨씬 더 좋을 것이라는 것을 기대할 수 있다.

5. 결 론

본 연구에서는 분산 환경에서 배열 데이터를 여러 호스트 상에 분할하여 생성하고, 이를 DSM 인터페이스로 사용할 수 있도록 지원하는 시스템인 JPAS을 설계하고 구현하였다. JPAS는 애플리케이션 프로그래밍에 사용되는 Java 클래스들과 실행 환경을 제공하는 2개의 Java 프로그램(BodyCreator와 BarrierManager)으로 구성된 Java 패키지 형태로서 추가적인 컴파일러나 런타임 시스템을 필요로 하지 않는다.

JPAS는 Java RMI를 이용하여 모든 공유 데이터를 복합 객체 형태의 Java 원격 객체로 생성시킨 후 각 호스트에서 데이터가 필요할 때는 핸들의 메소드를 통해 로컬 데이터처럼 사용하도록 한다. JPAS는 정수형과 실수형 배열을 공유 데이터 타입으로 사용한다. 표준 JDK의 컴파일러나 Java 가상 머신 대신에 type argument 기능을 제공하는 컴파일러나 가상 머신들을 이용하면 Java 클래스나 Java 인터페이스 등 사용자가 정의한 모든 데이터 타입을 배열의 원소로 사용할 수 있다.

JPAS는 데이터의 지역성 향상을 위해 캐쉬 읽기와 버퍼 갱신 기능을 제공한다. 버퍼와 캐쉬 외에도 사용자가 애플리케이션 특성적인 기능을 핸들의 메소드 형태로 추가하면 불필요한 통신 오버헤드를 조절할 수 있다. 실제 배열 데이터를 갖는 병렬 프로그램을 JPAS 패키지를 사용하여 작성한 후, 워크스테

이션 클러스터 상에서 실험해 본 결과, JPAS의 성능은 비교적 실용적임이 증명되었다.

참 고 문 헌

[1] A. Ferrari, "jPVM -The Java Parallel Virtual Machine," http://www.cs.virginia.edu/.

[2] V. Ivannikov, S. Gaissaryan, M. Donrachev, V. Etch and N. Shtaltovnaya, "DPJ:Java class library for development of data-parallel programs," http://www.isprus.ru/~dpj.

[3] "JavaPVM," http://www.isye.gatech.edu/chmsr/JavaPVM.

[4] P. Lu, "Aurora:Scoped Behavior for Per-Context Optimized Distributed Data Sharing," In *International Parallel Processing Symposium*, pp. 467-473, 1997.

[5] P. Lu, "Implementing Optimized Distributed Data Sharing Using Scoped Behaviour and a Class Library," In *3rd Conference on Object-Oriented Technologies and Systems, USENIX*, pp. 145-158, June 1997.

[6] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat: "An Efficient Implementation of Java's Remote Method Invocation," *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, May 1999, Atlanta, GA.

[7] Andrew C.Myers and Barbara Liskov, "Parameterized Types for Java," *Proceeding of the 24th ACM symposium on Principles of Programming Languages*, Paris, France, January 1997.

[8] M. Philippsen and M. Zenger, "JavaParty - Transparent Remote Objects in Java," In *ACM PPoPP Workshop on Java for Science and Engineering computation*, June 1997.

[9] Martin Odersky and Philip Wadler, "Pizza into Java: Translating theory into practice," *Proceeding of the 24th ACM symposium on Principles of Programming Languages*, Paris, France, January 1997.

[10] A. Yu, and W. Cox, "Java/DSM : A Platform for Heterogeneous Computing," In *ACM PPoPP Workshop on Java for Science and Engineering Computation*, June 1997.

[11] 황석찬, 최재영, 김명호, "Java 병렬 프로그래밍 환경(Java-based Parallel Programming Environment)," 한국정보과학회 병렬처리시스템 학술발표회 논문집, 9권, 1호, pp. 141-149, 1998.

임 혜 정

1997년 이화여자대학교 컴퓨터학과 학사.

1999년 이화여자대학교 컴퓨터학과 석사.

1999년~현재 KIST 트라이볼로지센터 위촉연구원.

관심분야 : 병렬처리, 운영체제, 컴퓨터 네트워크

김 명

1981년 이화여자대학교 수학과 학사.

1983년 서울대학교 계산통계학과 석사.

1993년 캘리포니아 주립대학교(산타바바라) 컴퓨터학과 박사.

1993년~1994년 캘리포니아 주립대학교(산타바바라) 컴퓨터학과 강사, Postdoc.

1995년~현재 이화여자대학교 컴퓨터학과 조교수.

관심분야 : 지식공학을 지원하는 병렬처리, 인터넷과 웹 기반 컴퓨팅.